Kernel Korner

The Devil's in the Details

This article, the third of five on writing character device drivers, introduces concepts of reading, writing, and using ioctl-calls.

by Georg v. Zezschwitz and Alessandro Rubini

Starting from the clean code environment of the two previous articles, we now turn to all the nasty interrupt stuff. Astonishingly, Linux hides most of this from us, so we do not need a single line of assembler...

Reading and writing

Right now, our magic `skel`-machine driver can load and even unload (painlessly, unlike in DOS), but it has neither read nor written a single character. So we will start fleshing out the `skel_read()` and `skel_write()` functions introduced in the previous article (under **fops and filp**). Both functions take four arguments:

```
Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count)
Static int skel_write (struct inode *inode,
                       struct file *file,
                       const char *buf,
                       int count)
```

The `inode` structure supplies the functions with information used already during the `skel_open()` call. For example, we determined from `inode->i_rdev` which board the user wants to open, and transferred this data--along with the board's base address and interrupt to the `private_data` entry of the file descriptor. We might ignore this information now, but if we did not use this hack, `inode` is our only chance to find out to which board we are talking.

The `file` structure contains data that is more valuable. You can explore all the elements in its definition in `<linux/fs.h>`. If you use the `private_data` entry, you find it here, and you should also make use of the `f_flags` entry--revealing to you, for instance, if the user wants blocking or non-blocking mode. (We explain this topic in more detail later on.)

The `buf` argument tells us where to put the bytes read (or where to find the bytes written) and `count` specifies how many bytes there are. But you must remember that every process has its own private address space. In kernel code, there is an address space common to all processes. When system calls execute on behalf of a specific process, they run in kernel address space, but are still able to access the user space. Historically, this was done through assembler code using the `fs` register; current Linux kernels hide the specific code within functions called `get_user_byte()` for reading a byte from user address space, `put_user_byte()` for writing one, and so on. They were formerly known as `get_fs_byte`, and only `memcpy_tofs()` and `memcpy_fromfs()` reveal these old days even on a DEC Alpha. If you want to explore, look in `<asm/segment.h>`.

Let us imagine ideal hardware that is always hungry to receive data, reads and writes quickly, and is accessed through a simple 8-bit data-port at the base address of our interface. Although this example is unrealistic, if you are impatient you might try the following code:

```
Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count) {
    int n = count;
    char *port = PORT0 ((struct Skel_Hw*)
                        (file->private_data));
    while (n--) {
        Wait till device is ready
        put_user_byte (inb_p (port), buf);
        buf++;
    }
    return count;
}
```

Notice the `inb_p()` function call, which is the actual I/O read from the hardware. You might decide to use its fast equivalent, `inb()`, which omits a minimal delay some slow hardware might need, but I prefer the safe way.

The equivalent `skel_write()` function is nearly the same. Just replace the `put_user_byte()` line by the following:

```
        outb_p (get_user_byte (buf), port);
```

However, these lines have a lot of disadvantages. What using them causes Linux to loop infinitely while waiting for a device that never becomes ready? Our driver should dedicate the time in the waiting loop to other processes, making use of all the resources in our expensive CPU, and it should have an input and output buffer for bytes arriving while we are not in `skel_read()` and corresponding `skel_write()` calls. It should also contain a time-out test in case of errors, and it should support blocking and non-blocking modes.

Blocking and Non-Blocking Modes

Imagine a process that reads 256 bytes at a time. Unfortunately, our input buffer is empty when `skel_read()` is called. So what should it do--return and say that there is no data yet, or wait until at least *some* bytes have arrived?

The answer is **both**. *Blocking* mode means the user wants the driver to wait till some bytes are read. *Non-blocking* mode means to return as soon as possible--just read all the bytes that are available. Similar rules apply to writing: *Blocking* mode means ``Don't return till you can accept some data," while *non-blocking* mode means: ``Return even if nothing is accepted." The `read()` and `write()` calls usually return the number of data bytes successfully read or written. If, however, the device is non-blocking and no bytes can be transferred, `-EAGAIN` is typically returned (meaning: *``Play it again, Sam"*). occasionally, old code may return `-EWOULDBLOCK`, which is the same as `-EAGAIN` under Linux.

Maybe now you are smiling as happily as I did when I first heard about these two modes. If these concepts are new for you, you might find the following hints helpful. Every device is opened by default in blocking mode, but you may choose non-blocking mode by setting the `O_NONBLOCK` flag in the `open()` call. You can even change the behaviour of your files later on with the `fcntl()` call. The `fcntl()`

call is an easy one, and the man page will be sufficient for any programmer.

Sleeping Beauty

Once upon a time, a beautiful princess was sent by a witch into a long, deep sleep, lasting for a hundred years. The world nearly forgot her and her castle, twined about by roses, until one day, a handsome prince came, kissed her, and awakened her --and all the other nice things happened that you hear about in fairy tales.

Our driver should do what the princess did while it is waiting for data: sleep, leaving the world spinning around. Linux provides a mechanism for that, called `interruptible_sleep_on()`. Every process reaching this call will fall asleep and contribute its time slices to the rest of the world. It will stay in this function till another process calls `wake_up_interruptible()`, and this ``prince'' usually takes the form of an interrupt handler that has successfully received or sent data, or Linux itself, if a time-out condition has occurred.

Installing an Interrupt Handler

The previous article in this series showed a minimal interrupt handler, which was called `skel_trial_fn()`, but its workings were not explained. Here, we introduce a ``complete'' interrupt handler, which will handle both input to and output from the actual hardware device. Figure 1 shows a simple version of its concept: When the driver is waiting for the device to get ready (blocking), it goes to sleep by calling `interruptible_sleep_on()`. A valid interrupt ends this sleep, restarting `skel_write()`.

Figure 1 does not include the double-nested loop structure we need when working with an internal output buffer. The reason is that if we can perform only writing within the `skel_write()` function there is no need for an internal output buffer. But our driver should catch data even while not in `skel_read()` and should write the data in the background even when not in `skel_write()`. Therefore, we will change the hardware writing in `skel_write()` to write to an output buffer and let the *interrupt handler* perform the real writing to the hardware. The interrupt and `skel_write()` will now be linked by the ``Sleeping Beauty'' mechanism and the output buffer.

The interrupt handler is installed and uninstalled during the `open()` and `close()` calls to the device, as suggested in the previous article. This task is handled by the following kernel calls:

```
#include <linux/sched.h>
int request_irq(unsigned int irq,
                void (*handler)
                     (int, struct pt_regs *),
                unsigned long flags,
                const char *device);
void free_irq(unsigned int irq);
```

The `handler` argument is the actual interrupt handler we wish to install. The role of the `flags` argument is to set a few features of the handler, the most important being its behaviour as a *fast* handler (`SA_INTERRUPT` is set in `flags`) or as a *slow* handler (`SA_INTERRUPT` is *not* set). A fast handler is run with all interrupts disabled, while a slow one is executed with all interrupts except itself enabled.

Finally, the `device` argument is used to identify the handler when looking at /proc/interrupts.

The handler function installed by `request_irq()` is passed only the interrupt number and the (often useless) contents of the processor registers.

Therefore, we'll first determine which board the calling interrupt belongs to. If we can't find any boards, a situation called a *spurious* interrupt has occurred, and we should ignore it. Typically interrupts are used to tell whether the device is ready either for reading *or* writing, so we have to find out by one or more hardware tests what the device wants us to do.

Of course, we should leave our interrupt handler quickly. Strangely enough, `printk()` (and thus the `PDEBUG` line) is allowed even within fast interrupt handlers. This is a very useful feature of the linux implementation. If you look at kernel/printk.c you'll discover that its implementation is based on wait queues, as the actual delivery of messages to log files is handled by an external process (usually klogd).

As shown in figure 2, Linux can handle a timeout when in `interruptible_sleep_on()`. For example, if you have are using a device to which you send an answer, and it is expected to reply within a limited time, causing a time-out to signal an I/O error (`-EIO`) in the return value to the user process might be a good choice.

Certainly the user process could care for this, too, using the alarm mechanism. But it is definitely easier to handle this in the driver itself. The timeout criteria is specified by `SKEL_TIMEOUT`, which is counted in *jiffies*. Jiffies are the steady heartbeat of a Linux system, a steady timer incremented every few milliseconds. The frequency, or number of jiffies per second, is defined by `HZ` in `<asm/param.h>` (included in `<linux/sched.h>`) and varies on different architectures (100 Hz Intel, 1 kHz Alpha). You simply have to set

```
#define SKEL_TIMEOUT timeout_seconds * HZ
/* ... */
current->timeout = jiffies + SKEL_TIMEOUT
```

and if `interruptible_sleep_on` timed out, `current->timeout` will be cleared after return.

Be aware that interrupts might happen within `skel_read()` and `skel_write()`. Variables that might be changed within the interrupt should be declared as `volatile`. They also need to be protected to avoid race conditions. The classic code sequence to protect a critical region is the following:

```
unsigned long flags;
save_flags (flags);
cli ();
critical region
restore_flags (flags);
```

Finally, the code for the ``complete" error handler:

```
#define SKEL_IBUFSIZ 512
#define SKEL_OBUFSIZ 512
/* for 5 seconds timeout */
#define SKEL_TIMEOUT (5*HZ)

/* This should be inserted in the Skel_Hw-structure */
typedef struct Skel_Hw {
    /* write position in input-buffer */
    volatile int ibuf_wpos;
    /* read position in input-buffer */
```

```
    int ibuf_rpos;
    /* the input-buffer itself */
    char *ibuf;
    /* write position in output-buffer */
    int obuf_wpos;
    /* read position in output-buffer */
    volatile int buf_rpos;
    char *obuf;
    struct wait_queue *skel_wait_iq;
    struct wait_queue *skel_wait_oq;
    [...]
}

#define SKEL_IBUF_EMPTY(b) \
 ((b)->ibuf_rpos==(b)->ibuf_wpos)
#define SKEL_OBUF_EMPTY(b) \
 ((b)->obuf_rpos==(b)->obuf_wpos)
#define SKEL_IBUF_FULL(b) \
 (((b)->ibuf_wpos+1)%SKEL_IBUFSIZ==(b)->ibuf_rpos)
#define SKEL_OBUF_FULL(b) \
 (((b)->obuf_wpos+1)%SKEL_OBUFSIZ==(b)->obuf_rpos)

Static int skel_open (struct inode *inode,
                      struct file *filp) {
    /* .... */
    /* First we allocate the buffers */
    board->ibuf = (char*) kmalloc (SKEL_IBUFSIZ,
                                   GFP_KERNEL);
    if (board->ibuf == NULL)
        return -ENOMEM;
    board->obuf = (char*) kmalloc (SKEL_OBUFSIZ,
                                   GFP_KERNEL);
    if (board->obuf == NULL) {
        kfree_s (board->ibuf, SKEL_IBUFSIZ);
        return -ENOMEM;
    }
    /* Now we clear them */
    ibuf_wpos = ibuf_rpos = 0;
    obuf_wpos = obuf_rpos = 0;
    board->irq = board->hwirq;
    if ((err=request_irq(board->irq>
                         skel_interrupt,
                         SA_INTERRUPT, "skel")))
        return err;
}

Static void skel_interrupt(int irq,
                    struct pt_regs *unused) {
    int i;
    Skel_Hw *board;

    for (i=0, board=skel_hw; i<skel_boards;
         board++, i++)
         /* spurious */
        if (board->irq==irq) break;
    if (i==skel_boards) return;
    if (board_is_ready_for_input)
        skel_hw_write (board);
    if (board_is_ready_for_output)
        skel_hw_read (board);
```

```
}

Static inline void skel_hw_write (Skel_Hw *board){
    int rpos;

    char c;
    while (! SKEL_OBUF_EMPTY (board) &&
        board_ready_for_writing) {
        c = board->obuf [board->obuf_rpos++];
        write_byte_c_to_device
        board->obuf_rpos %= SKEL_OBUF_SIZ;
    }
    /* Sleeping Beauty */
    wake_up_interruptible (board->skel_wait_oq);
}

Static inline void skel_hw_read (Skel_Hw *board) {
    char c;

    /* If space left in the input buffer & device ready: */
    while (! SKEL_IBUF_FULL (board) &&
        board_ready_for_reading) {
        read_byte_c_from_device
        board->ibuf [board->ibuf_wpos++] = c;
        board->ibuf_wpos %= SKEL_IBUFSIZ;
    }
    wake_up_interruptible (board->skel_wait_iq);
}

Static int skel_write (struct inode *inode,
                       struct file *file,
                       char *buf, int count) {
    int n;
    int written=0;
    Skel_Hw board =
        (Skel_Hw*) (file->private_data);

    for (;;) {
        while (written<count &&
               ! SKEL_OBUF_FULL (board)) {
            board->obuf [board->obuf_wpos] =
                get_user_byte (buf);
            buf++; board->obuf_wpos++;
            written++;
            board->obuf_wpos %= SKEL_OBUFSIZ;
        }
        if (written) return written;
        if (file->f_flags & O_NONBLOCK)
            return -EAGAIN;
        current->timeout = jiffies + SKEL_TIMEOUT;
        interruptible_sleep_on (
            &(board->skel_wait_oq));
        /* Why did we return? */
        if (current->signal & ~current->blocked)
        /* If the signal is not not being
           blocked */
            return -ERESTARTSYS;
        if (!current->timeout)
            /* no write till timout: i/o-error */
            return -EIO;
```

```
    }
}

Static int skel_read (struct inode *inode,
                      struct file *file,
                      char *buf, int count) {
    Skel_Hw board =
        (Skel_Hw*) (file->private_data);
    int bytes_read = 0;

    if (!count) return 0;

    if (SKEL_IBUF_EMPTY (board)) {
        if (file->f_flags & O_NONBLOCK)
            /* Non-blocking */
            return -EAGAIN;

        current->time_out = jiffies+SKEL_TIMEOUT;
        for (;;) {
            skel_tell_hw_we_ask_for_data
            interruptible_sleep_on (
                &(board->skel_wait_iq));
            if (current->signal
                & ~current->blocked)
                return -ERESTARTSYS;
            if (! SKEL_IBUF_EMPTY (board))
                break;
            if (!current->timeout)
                /* Got timeout: return -EIO */
                return -EIO;
        }
    }
    /* if some bytes are here, return them */

    while (! SKEL_IBUF_EMPTY (board)) {
        put_user_byte (board->ibuf
                          [board->ibuf_rpos],
                      buf);
        buf++; board->ibuf_rpos++;
        bytes_read++;
        board->ibuf_rpos %= SKEL_IBUFSIZ;
        if (--count == 0) break;
    }
    if (count) /* still looking for some bytes */
        skel_tell_hw_we_ask_for_data
    return bytes_read;
}
```

Handling `select()`

The last important I/O function to be shown is `select()`, one of the most interesting parts of Unix, in our opinion.

The `select()` call is used to wait for a device to become ready, and is one of the most scary functions for the novice C programmer. While its use from within an application is not shown here, the driver-specific part of the system call is shown, and its most impressive feature is its compactness.

Here's the full code:

```
Static int skel_select(struct inode *inode,
                       struct file *file,
                       int sel_type,
                       select_table *wait) {
    Skel_Clientdata *data=filp->private_data;
    Skel_Board *board=data->board;
    if (sel_type==SEL_IN) {
        if (! SKEL_IBUF_EMPTY (board))
            /* readable */
            return 1;
        skel_tell_hw_we_ask_for_data;
        select_wait(&(hwp->skel_wait_iq), wait);
        /* not readable */
        return 0;
    }
    if (sel_type==SEL_OUT) {
        if (! SKEL_OBUF_FULL (board))
            return 1;  /* writable */
        /* hw knows */
        select_wait (&(hwp->skel_wait_oq), wait);
        return 0;
    }
    /* exception condition: cannot happen */
    return 0;
}
```

As you can see, the kernel takes care of the hassle of managing wait queues, and you have only to check for readiness.

When we first wrote a `select()` call for a driver, we didn't understand the `wait_queue` implementation, and you don't need to either. You only have to know that the code works. `wait_queues` *are* challenging, and usually when you write a driver you have no time to accept the challenge.

Actually, `select` is better understood in its relationships with read and write: if `select()` says that the file is readable, the next read must not block (independently of `O_NONBLOCK`), and this means you have to tell the hardware to return data. The interrupt will collect data, and awaken the queue. If the user is selecting for writing, the situation is similar: the driver must tell if `write()` will block or not. If the buffer is full it will block, but you don't need to tell the hardware about it, since `write()` has already told it (when it filled the buffer). If the buffer is not full, the write won't block, so you return 1.

This way to think of selecting for write may appear strange, as there are times when you need to write synchronously, and you may expect that a device is writable when it has already accepted pending input. Unfortunately, this way of doing things will break the blocking/nonblocking machinery, and thus an extra call is provided: if you need to write synchronously, the driver must offer (within its `fops`) the `fsync()` call. The application invokes `fops->fsync` through the `fsync()` system call, and if the driver doesn't support it, `-EINVAL` is returned.

`ioctl()`—Passing Control Information

Imagine that you want to change the baud-rate of a serial multiport card you have built. Or tell your frame grabber to change the resolution of an image. Or whatever else... You could wrap these instructions into a series of escape sequences, such as, for example, the screen positioning in ANSI

emulation. But, the normal method for this is to make an `ioctl()` call.

`ioctl()` calls as defined in `<sys/ioctl.h>` have the form

```
ioctl (int file_handle, int command, ...)
```

where ... is considered to be one argument of the type `char *` (according to the `ioctl` man page). Strange as it may be, the kernel receives these arguments in `fs/ioctl.c` in the form:

```
int sys_ioctl (unsigned int fd, unsigned int cmd,
               unsigned long arg);
```

To add to the confusion, `<linux/ioctl.h>` gives detailed rules how the commands in the second parameter should be built, but nobody in all the drivers is actually following these ideas yet.

In any case, rather than cleaning up the Linux source tree, let's concentrate on the general *idea* of `ioctl()` calls. As the user, you pass the file handle and a command in the first two arguments and pass as the third parameter a pointer to a data structure the driver should read and/or write.

A few commands are interpreted by the kernel itself--for example, `FIONBIO`, which changes the blocking/non-blocking flag of the file. The rest is passed to our own, driver-specific `ioctl()` call, and arrives in the form:

```
int skel_ioctl (struct inode *inode,
                struct file *file,
                unsigned int cmd,
                unsigned long arg)
```

Before we show a small example of a `skel_ioctl()` implementation, the commands you define should obey the following rules:

1. Pick up a free MAGIC number from /usr/src/linux/MAGIC and make this number the upper eight bits of the 16-bit command word.
2. Enumerate commands in the lower eight bits.

Why this? Imagine ``Silly Billy'' starts his favorite terminal program minicom to connect to his mailbox. ``Silly Billy'' accidentally changed the serial line minicom uses from /dev/ttyS0 to /dev/skel0 (he is quite silly). The next thing minicom does is initialize the ``serial line'' with an `ioctl()` using `TCGETA` as command. Unfortunately, your device driver, hidden behind /dev/skel0, uses that number to control the voltage for a long-term experiment in the lab...

If the upper eight bits in the commands for `ioctl()` differ from driver to driver, every `ioctl()` to an inappropriate device will result in an `-EINVAL` return, protecting us from extremely unexpected results.

Now, to finish this section, we will implement an `ioctl()` call reading or changing the timeout delay in our driver. If you want to use it, you have to introduce a new variable

```
unsigned long skel_timeout = SKEL_TIMEOUT;
```

right after the definition of `SKEL_TIMEOUT` and replace every later occurrence of `SKEL_TIMEOUT` with `skel_timeout`.

We choose the `MAGIC '4'` (the ASCII character 4) and define two commands:

```
# define SKEL_GET_TIMEOUT 0x3401
# define SKEL_SET_TIMEOUT 0x3402
```

In our user process, these lines will double the time-out value:

```
/* ... */
unsigned long timeout;
if (ioctl (skel_hd, SKEL_GET_TIMEOUT,
           &timeout) < 0) {
    /* an error occurred (Silly billy?) */
    /* ... */
}
timeout *= 2;
if (ioctl (skel_hd, SKEL_SET_TIMEOUT,
           &timeout) < 0) {
    /* another error */
    /* ... */
}
```

And in our driver, these lines will do the work:

```
int skel_ioctl (struct inode *inode,
                struct file *file,
                unsigned int cmd,
                unsigned long arg) {
    switch (cmd) {
    case SKEL_GET_TIMEOUT:
        put_user_long(skel_timeout, (long*) arg);
        return 0;
    case SKEL_SET_TIMEOUT:
        skel_timeout = get_user_long((long*) arg);
        return 0;
    default:
        return -EINVAL; /* for Silly Billy */
    }
}
```

Georg and Alessandro are both 27-year-old Linuxers with a taste for the practical side of Computer Science and a tendency to avoid sleep.

[| Magazine Table of Contents]

[Magazine Table of Contents]